Streaming Computation of Delaunay Triangulations

Martin Isenburg
University of California
at Berkeley

Yuanxin Liu
University of North Carolina
at Chapel Hill

Jonathan Shewchuk
University of California
at Berkeley

Jack Snoeyink
University of North Carolina
at Chapel Hill

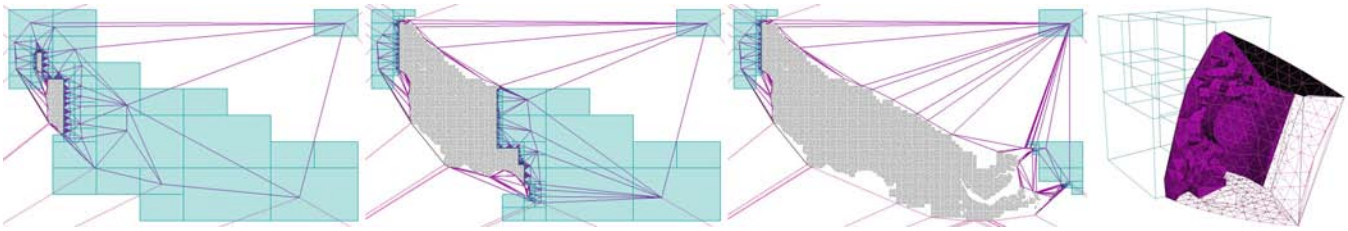

Figure 1: Streaming computation of Delaunay triangulations in 2D (Neuse River) and 3D. Blue quadrants or octants are unfinalized space where future points will arrive. Purple triangles and tetrahedra are in memory. Black points and their triangles and tetrahedra have already been written to disk or piped to the next application.

Abstract

We show how to greatly accelerate algorithms that compute Delaunay triangulations of huge, well-distributed point sets in 2D and 3D by exploiting the natural spatial coherence in a stream of points. We achieve large performance gains by introducing *spatial finalization* into point streams: we partition space into regions, and augment a stream of input points with finalization tags that indicate when a point is the last in its region. By extending an incremental algorithm for Delaunay triangulation to use finalization tags and produce streaming mesh output, we compute a billion-triangle terrain representation for the Neuse River system from 11.2 GB of LIDAR data in 48 minutes using only 70 MB of memory on a laptop with two hard drives. This is a factor of twelve faster than the previous fastest out-of-core Delaunay triangulation software.

CR Categories: I.3.5 [COMPUTER GRAPHICS]: Computational Geometry and Object Modeling—Geometric algorithms

Keywords: geometry processing, Delaunay triangulation, stream processing, TIN terrain model, spatial finalization

1 Introduction

New instruments have made huge geometric data sets common in terrain modeling (LIDAR, synthetic aperture radar), medical image analysis (magnetic resonance imaging, tomography), and computer-aided engineering (laser range scanning, finite element methods). These data sets are often many times larger than the memories of commodity computers, and overwhelm the algorithms and data formats used to manage and analyze them. Our expanding capacity to collect geometric data has inspired a recent burst of research on streaming representations of large-scale geometry [Isenburg et al. 2003; Isenburg and Lindstrom 2005; Pajarola 2005].

We detail here how we use streaming computation to construct a billion-triangle Delaunay triangulation of a planar point set in 48 minutes with an off-the-shelf laptop computer plus a firewire drive, using 70 MB of memory to produce a 16.9 GB triangulation. This is about a factor of twelve faster than the previous best out-of-core Delaunay triangulator, by Agarwal, Arge, and Yi [2005]; see Section 6. We also construct a nine-billion-triangle, 152 GB triangulation in under seven hours using 166 MB of main memory.

A *streaming* computation makes a small number of sequential passes over a data file (ideally, one pass), and processes the data using a memory buffer whose size is a fraction of the stream length. We have implemented two- and three-dimensional triangulators that read streams of points as input, and produce Delaunay triangulations in streaming mesh formats. The memory footprint of the 2D triangulator is typically less than 0.5% of the output mesh size (sometimes much less). The memory footprint of the 3D triangulator is typically less than 10% of the output mesh size when the points are roughly uniformly distributed in a volume.

The main new idea in our streaming Delaunay triangulators is *spatial finalization* (which differs from the *topological finalization* of mesh entities like points and triangles in previous papers). We partition space into regions, and include *finalization tags* in the stream that indicate when no more points in the stream will fall in specified regions. Our triangulators certify triangles or tetrahedra as Delaunay when the finalization tags show it is safe to do so. This make it possible to write them out early, freeing up memory to read more from the input stream. Because only the unfinalized parts of a triangulation are resident in memory, the memory footprint remains small. We created our triangulators by making modest changes to existing incremental Delaunay triangulation implementations—no new triangulation algorithm was needed.

Streaming algorithms can succeed only if streams have sufficient *spatial coherence*—a correlation between the proximity in space of geometric entities and the proximity of their representations in the stream. We present evidence in Section 3 that huge real-world data sets often do have sufficient spatial coherence. This is not surprising; if they didn't, the programs that created them would have bogged down due to thrashing. Moreover, we can add more spatial coherence to a stream by *chunking*—reordering points (in memory, without resorting to an external sort) so that all the points in a region appear consecutively. Many external memory algorithms sort the geometry as a first step. One of our contributions is the observation that spatial coherence often enables us to triangulate a large point set in the time it takes just to sort it. (See Section 6.)

With these ideas and a laptop, we can process the 11.2 GB of bare-earth LIDAR data for the Neuse River Basin, comprising over 500 million points (double-precision x, y, and height coordinates). This data comes from the NC Floodplain Mapping project[1], begun after Hurricane Floyd in 1999. North Carolina was the first state to

for demo software & source code see http://www.cs.unc.edu/∼isenburg/sd/

[1] http://www.ncfloodmaps.com

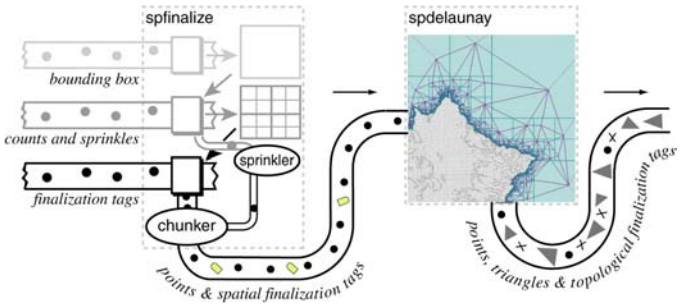

Figure 2: The finalizer reads the points thrice and pipes a spatially finalized point stream to the triangulator, which writes out a topologically finalized streaming mesh.

use LIDAR (Light Detection and Ranging, an airborne laser scanning technology) and capture elevation points to assess flood risks, set insurance premiums, and create disaster plans for an entire state. The sheer enormity of the models has hindered their processing, delaying the project's completion from 2002 to 2007 [Quillin 2002].

Faced with a half billion points, a typical in-core algorithm, with perhaps a gigabyte at its disposal, must resort to virtual memory. Then computations like following pointers through linked lists or triangulation data structures, maintaining priority queues, and allocating and freeing objects produce memory access patterns that cause thrashing—excessive paging—slowing execution to a crawl.

We triangulate huge point sets with two concurrent programs depicted in Figure 2. The *finalizer* reads a stream of raw points three times from disk. During the first pass it finds the bounding box, on which we overlay a grid of rectangular regions. During the second pass it counts the number of points in each region. During the third pass it inserts spatial finalization tags, reorders the points, and writes a spatially finalized point stream to a pipe. Two finalizer components reorder points during the third pass: the *chunker* delays points so that the points in each region are contiguous, which improves spatial coherence, and the *sprinkler* promotes representative points (sampled during the second pass) to earlier positions in the stream, which averts the risk of quadratic running time. The triangulator reads the finalized point stream from the pipe and triangulates it with an incremental Delaunay algorithm, writing a finalized mesh stream while still reading the finalized point stream.

The two programs triangulate the 11.2 GB Neuse River Basin point stream, producing a 16.9 GB mesh, in 48 minutes using 70 MB of memory. The finalizer occupies 60 MB of memory (used mainly to reorder points), and the triangulator occupies 10 MB—less than 0.1% of the size of the mesh. If the triangulator can read an already-finalized point stream from disk, there is no need for the finalizer, and the triangulator runs in 35 minutes.

The triangulation may be piped directly to another application—for instance, software for mesh simplification, or for extracting contour lines or drainage networks from terrain. Because stream processing modules typically have small memory footprints, we run chains of them concurrently and stream gigabytes through them. A major benefit of streaming (*without* sorting the points as a first step) is quick feedback: For example, a user can pipe the triangulator's output to our streaming isocontour extraction module, whose output is piped to a visualization module. Isocontours begin to appear within minutes (or seconds), as our processes produce output while still consuming input. If they look wrong, the user can about the pipeline and restart all the streaming components with different parameters. With other methods, users must wait hours for the triangulation to finish before glimpsing the results.

We advocate that applications that create huge geometric data sets, such as scientific simulations, should strive to write their output in the form of spatially coherent, spatially finalized, streaming geometry. The effort needed to do so is often small, and the reward is the ability to perform large-scale computations normally thought to be the exclusive domain of parallel supercomputers.

Unfortunately, our streaming triangulators do not enjoy the same out-of-core performance for surface point clouds in 3D as they do for terrains and volume-filling point clouds. The difficulty is caused by the many large circumspheres in the Delaunay triangulations of surface point clouds, which thwart spatial finalization from certifying tetrahedra. We believe a more sophisticated finalization technique can overcome this hurdle. See the Conclusions for details.

2 Processing large geometric data sets

How can we handle large data sets? Powerful computers with large memories suffice for those who have them (and are often responsible for producing the data sets). To make large data sets useful to the wider audience that have commodity processors, however, we need algorithms that use a small amount of memory wisely. Here we review general approaches to algorithms for large geometric data sets, and the literature on computing large Delaunay triangulations.

2.1 Algorithms for large data sets

Several types of algorithms are used to process large geometric data sets: *divide-and-conquer algorithms*, which cut a problem into small subproblems that can be solved independently; *cache-efficient algorithms*, which cooperate with the hardware's memory hierarchy (caches and virtual memory); *external memory algorithms*, which exercise control over where, when, and how data structures are stored on disk (rather than trusting the virtual memory); and *streaming algorithms*, which sequentially read a stream of data (usually once, perhaps in several passes) and retain only a small portion of the information in memory. All of these algorithms try to exploit or create spatial coherence.

Divide-and-conquer algorithms for huge data sets are, for some problems, difficult to design: they often require ingenious algorithms to choose the cuts, necessitate tedious programming to communicate across the cuts, or suffer from poor-quality results near the cuts. For Delaunay triangulations, the very act of choosing cuts so that no further communication is needed requires a convex hull algorithm that itself can process huge data sets [Blelloch et al. 1999].

Cache-efficient algorithms (which often also cooperate well with virtual memory) fall into two categories. Some software is optimized for a particular cache architecture—a well-known example is BLAS (the Basic Linear Algebra Subprograms), optimized by most microprocessor vendors for their architectures. Some software is *cache-oblivious*, designed to cooperate well with any cache or virtual memory, regardless of the details of its architecture. This category includes heuristics for cache-oblivious data layouts that do well in practice [Yoon et al. 2005], and cache-oblivious algorithms that offer guaranteed bounds on the amount of traffic between cache and memory [Kumar 2003] (and sometimes do well in practice).

External memory algorithms use disks for temporary storage of data structures that do not fit in memory, and explicitly control data movement and data layout on disk with the goal of minimizing the number of disk accesses [Vitter 2001]. Like cache-oblivious algorithms, external memory algorithms have received a lot of attention from theoreticians, who give provable bounds on the number of disk accesses their algorithms perform. Most of these algorithms build sophisticated data structures on disk, notably B-trees.

Streaming, the approach we advocate here, differs from external memory algorithms in that nothing is temporarily paged out to external memory. The disk is used only for input and output. The algorithm makes one or a few sequential passes over the data and restricts computations to those parts that are in memory. As it can neither backtrack nor store more than a small fraction of the stream, it needs a mechanism to decide what parts of the data to retain and for how long, and where it can safely complete computations.

Some online algorithms can be remarkably more effective if the stream includes a small amount of information about the "future" of the stream [Karp 1992]. Representations for streaming

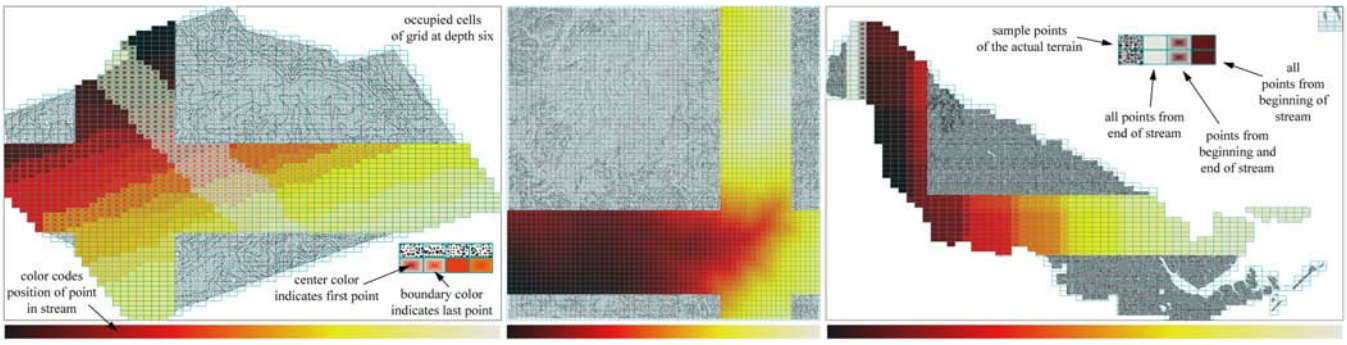

Figure 3: Three terrain data sets: the 6 million-point "grbm" (LIDAR data of Baisman Run at Broadmoor, Maryland, left), the 67 million-point "puget" (middle) and the 0.5 billion-point "neuse" data set (right). Colors illustrate spatial coherence in selected grid cells: each cell's center is colored by the arrival time of its first point in the stream, and each cell's boundary is colored by the arrival time of its last point in the stream, with time increasing from black to white along the color ramp (bottom).

meshes [Isenburg and Gumhold 2003; Isenburg et al. 2003; Isenburg and Lindstrom 2005] contain not only points, triangles, and tetrahedra, but also *finalization* tags that certify when a topological entity is seen for the last time. Finalization tags permit a geometric algorithm to output partial results and discard associated information, freeing room in memory for more data to stream in.

2.2 Delaunay triangulations and large data sets

The Delaunay triangulation and its dual Voronoi diagram [Okabe et al. 2000] have been ubiquitous in geometry processing since algorithms for them first appeared in the 1970s [Frederick et al. 1970; Shamos and Hoey 1975]. The Delaunay triangulation (or tetrahedralization) of a set of points has the property that the circumscribing circle of every triangle, or the circumscribing sphere of every tetrahedron, encloses no point in the set. Many surveys of Delaunay triangulations are available: see Fortune [1992] for mathematical properties, Su and Drysdale [1995] for a summary of two-dimensional algorithms and their behavior in practice, and Liu and Snoeyink [2005] for a survey and comparison of five three-dimensional implementations.

Because of its simplicity, we implemented Lawson's [1977] incremental insertion algorithm as modified and extended to any dimension by Bowyer [1981] and Watson [1981]. Clarkson and Shor [1989] were first to show that incremental algorithms can run in optimal time in any dimension if the points are inserted in random order. Nearly all modern three-dimensional implementations use incremental insertion, with various strategies for *point location*—determining where each new point should be inserted.

Our 2D in-core standard for comparison is the divide-and-conquer algorithm [Shamos and Hoey 1975] as implemented in Triangle [Shewchuk 1996], which runs in optimal $O(n \log n)$ time and is the fastest in practice (if the data structures fit in main memory).

Recent papers address the problem of computing Delaunay triangulations too large to fit in memory. Blandford et al. [2005] describe data structures for dynamically maintaining compressed triangulations in two or three dimensions, thereby increasing the size of triangulation that fits in memory by a factor of three to five.

For larger triangulations, researchers turn to disk storage. Unfortunately, the randomness that makes incremental insertion fast distributes data structures randomly through memory, with no spatial coherence, so the virtual memory thrashes as soon as the physical memory is exhausted. Amenta, Choi, and Rote [2003] address this problem (in any dimension) by choosing a point insertion order that has strong spatial coherence, but retains just enough randomness to preserve the proof of optimal running time. They call this order a *biased randomized insertion order* (BRIO). By using a BRIO, they increase substantially the size of triangulation they can construct with a fixed main memory and a large virtual memory.

As an unexpected bonus, their method speeds up point location so much that their implementation triangulates most real-world three-dimensional point sets in linear time, after an $O(n \log n)$ sorting step whose hidden constant factor is tiny. Buchin [2005] proves that incremental insertion, coupled with similar randomization and point location based on space-filling curves and bucketing, runs in $O(n)$ time on certain random point sets.

Agarwal, Arge, and Yi [2005] have designed and implemented an external memory algorithm for constructing constrained Delaunay triangulations in the plane, with theoretical bounds on the number of disk block accesses their algorithm performs. They use a divide-and-conquer approach in which a small random sample of the points breaks the problem up into small subproblems, which are triangulated in-core by Triangle. Their algorithm uses no complicated external data structures (not even B-trees) and is akin to streaming, but it does many read passes over the points. Our streaming implementation outperforms their external memory implementation strikingly—see Section 6—but we have not yet implemented support for *constrained* Delaunay triangulations.

3 Point streams with spatial finalization

Isenburg and Lindstrom [2005] observe that large mesh data files have inherent *topological* coherence (i.e., locality in vertex references). It comes as no surprise that geometric data sets also exhibit inherent *spatial* coherence (i.e., locality in vertex positions). Figure 3 illustrates the spatial coherence in three terrain data sets. We overlay the points with a grid, and color selected cells according to the time of arrival (in the data stream) of their first point (inner color) and their last point (outer color). A cell with two contrasting colors has a large gap between the first and last points. Fortunately, the proportion of cells with widely contrasting colors is small, indicating that these data sets have good spatial coherence.

The aim of a streaming format is to document coherence so that algorithms can exploit it. Isenburg and Lindstrom describe a *streaming mesh* format consisting of vertices, triangles, and *finalization tags*. A finalization tag *finalizes* a vertex v after the last triangle incident on v appears in the stream. The tag tells the application processing the streaming mesh that it may complete any local computation that was waiting for v's local topology, output partial results, and free some data structures (probably including the one representing v). We call this *topological finalization*, because it depends purely on the connectivity of the mesh.

There is no topology in a stream of points, but one can define other notions of finalization suited to algorithms that operate on point sets. Pajarola [2005] globally sorts points along one axis to derive what we call *k-neighbor finalization* from the sorted point stream: a point is finalized after its k nearest neighbors have arrived. We advocate a more general notion of *spatial finalization* that provides similar spatial guarantees for stream processing of points without imposing a strict global order on the points.

We subdivide space into regions, and finalize a region after the

last point in that region arrives. Finalization injects information about the future into a stream—in this case, promises that certain regions contain no additional points. These promises can be used to certify that a Delaunay circumsphere is empty, and could perhaps be used in surface reconstruction to certify that all points within a local neighborhood are known, or in elevation map generation to certify that all points needed to rasterize a tile have arrived.

Formally, we define a *point stream* to be a sequence of points, and a *spatially finalized point stream* (or a *finalized point stream* or simply *finalized points*) to be a sequence of entities, of which all but one are points or finalization tags. The first entity in the stream is special: it specifies a subdivision of space into regions, such that each point in the stream lies in one of the regions. A point is *finalized* when a spatial finalization tag arrives for a region containing the point. The *width* of a finalized point stream is the maximum number of unfinalized points at any time. A stream's width is usually a lower bound on the number of points a point processing algorithm must retain in a buffer at any one time.

Geometric processing tasks whose operations are sufficiently "local" in space can take advantage of spatial finalization to process a point stream in a memory footprint roughly proportional to the width of the stream, rather than its length. Most large real-world point sets have enough spatial coherence to be streamed with low width, but they do not document this fact with spatial finalization tags. We add them with a program called a *finalizer*, which makes three passes over an input point stream and outputs a spatially finalized point stream. Normally, the output is piped directly to an application program—in particular, our Delaunay triangulators—so there is no need to store the finalized point stream on disk.

Our choice of partition is a rectangular $2^k \times 2^k$ grid of cells. The finalizer stores cells in a hash table, so only cells that contain points take up memory. The finalizer's first pass over the input stream simply computes the points' smallest axis-parallel bounding box. This box is partitioned into equal cells, and the second pass counts how many points fall into each cell. The third pass is like the second pass, except that the finalizer decrements these counters instead of incrementing them. When a cell's counter reaches zero, the finalizer inserts a finalization tag for that cell into the stream.

Although real-world point sets usually have a lot of spatial coherence, it pays to add more. A finalizer can do more than just finalize points—ours (in its default setting) also reorders them. During the third pass, it buffers all the points in each cell until the cell's counter reaches zero, then it releases all the cell's points into the output stream, followed by the cell's finalization tag. We call this act *chunking*. Chunking reduces width and increases spatial coherence. It exploits and enhances the coherence already in the data, but requires far less work than fully sorting the points.

The buffers used for chunking increase the memory footprint of the finalizer, but that increase is more than offset by the reduced memory footprint of the stream-based application receiving the finalizer's output stream. Buffers filled with points take much less memory per point than the corresponding triangulation data structure. We implement the point buffers as linked lists of memory page-sized blocks, so that if there is insufficient main memory we use virtual memory efficiently. However, all the data sets we used in this paper were coherent enough to chunk in main memory.

Sometimes it pays to give up a little spatial coherence. Recall from Section 2.2 that several Delaunay triangulation algorithms use random sampling to improve their performance. Adding randomness to the input order is not just a technique for getting theoretical results. If a Delaunay triangulator inserts the vertices of a large square grid in Cartesian order—a natural order for stream processing—it will degenerate to quadratic running time.

Our streaming implementation borrows ideas from Amenta et al. [2003] to avoid this danger. The finalizer samples a small fraction of points from the stream, and promotes them to earlier spots in the stream. On a local level, the finalizer reorders the points within each cell into a BRIO as a part of chunking. When the last point in a cell arrives in the input stream, the finalizer moves a sample of randomly selected points to the front of the chunk before releasing them into the output stream, thereby averting the possibility of a quadratic number of changes to the triangulation within the cell.

input points	k	time/pass, m:ss 1	2	3	MB	points buffered	occupied cells	points per cell avg	max
puget	6	0:27	0:27	0:41	25	2,018,478	4,096	16,387	41,681
67M pts	7	0:27	0:27	0:39	16	1,101,576	16,384	4,097	12,103
768 MB	8	0:27	0:27	0:40	13	653,058	65,536	1,024	3,290
neuse	8	5:55	5:56	9:12	93	3,842,202	19,892	25,142	66,171
500M pts	9	5:55	5:54	8:23	60	2,249,268	77,721	6,435	20,208
11.2 GB	10	5:55	5:55	8:05	59	1,396,836	306,334	1,633	6,544
neuse 3×3	10	52:58	53:18	– : –	136	4,617,984	314,797	14,299	41,135
4.5B pts	11	52:58	53:04	– : –	155	2,169,216	1,234,615	3,645	13,430
110 GB	12	52:58	53:20	– : –	425	978,390	4,880,173	922	4,267

Table 1: Running times (minutes:seconds) and maximum memory footprints (MB) for the three passes of `spfinalize` when finalizing three terrain point sets using $2^k \times 2^k$ grids. Each pass reads raw points from a firewire drive, and the third pass simultaneously writes finalized points to the local disk. We also report the maximum number of points buffered in memory at one time, the number of occupied grid cells, and the average and maximum points per occupied cell. Third pass timings are omitted for the "neuse" tiling, because we cannot store 110 GB to disk; but see Table 2.

To be fully effective, points must also be sampled and promoted globally. During its second pass over the input, our finalizer builds a quadtree whose leaves are the grid cells, and stores one point from each quadrant (biased towards points near the quadrant's center) at each level of the quadtree. During the third pass, it moves these *sprinkle points* to early spots in the output stream. In the manner of Amenta et al., we could write them out in level order (from the top to the bottom level of the quadtree) at the front of the stream, but the number of points we reorder is so large that the width of the stream would blow up. Instead of releasing them all at once, we "sprinkle" them into the stream in a lazy fashion: when a cell is finalized, the sprinkle points associated with all its ancestors and their immediate children in the quadtree are released (if they haven't already been) before the points of the finalized cell are released.

Table 1 documents the time and memory requirements for spatial finalization of the largest two point sets depicted in Figure 3. The 67 million points of "puget" are the vertices of an unstructured TIN model of the Puget Sound, generated by Yoon et al. [2005] through adaptive simplification of a regular triangulation derived from a USGS digital elevation map. The "neuse" point set is described in Section 1. The "neuse 3×3" point set is nine tiles of "neuse" arranged in a non-overlapping grid. The points in all three sets are distributed fairly uniformly—the maximum number of points in a grid cell is a small multiple of the average.

From the timings we see that the first two passes are strictly I/O-bound—our LaCie 5,400 RPM firewire drive has a throughput of 2 GB/min, so it takes six minutes to read the 11.2 GB "neuse" point set in each pass. The third pass is also I/O-bound, but it is slower because it simultaneously writes finalized points chunk-by-chunk to the local disk. If we discard the output instead, the third pass is almost as fast as the first two passes, with some additional time spent chunking the points. A finer grid (a larger k) means that points are finalized more frequently and in smaller chunks. This reduces the maximum number of points buffered for chunking, but increases the memory occupied by the quadtree data structure and the sprinkles. Of the 425 MB of memory used to finalize "neuse 3×3" with a $2^{12} \times 2^{12}$ grid, the quadtree and sprinkles take 400 MB.

An observant reader might object that a point-creating application could destroy the coherence that our finalizer is expecting simply by delaying one point in each cell to the end of the stream. Indeed, the "grbm" data set makes the finalizer buffer many points, because there is a diagonal stripe across the terrain at the end of the file. (It appears that the airplane was still collecting data on the way home.) This vulnerability is not an inherent limitation of stream-

ing, only of our current implementation of the finalizer. Although we did not find it necessary with our current data sets, we could reorder such points by identifying them during the second pass, and storing them in a memory buffer or a temporary file. If there are too many points to buffer, then the data set is spatially incoherent, and any out-of-core triangulator must globally rearrange the data.

4 Streaming 2D Delaunay triangulation

Conventional Delaunay triangulation programs output triangles after all the input points have been processed. By taking as input a spatially finalized point stream, our triangulator `spdelaunay2d` constructs a Delaunay triangulation incrementally and outputs a triangle whenever it determines that the triangle is *final*—that its circumcircle does not touch or enclose an unfinalized cell. Such a triangle must be in the Delaunay triangulation, since no point arriving in the future can be inside the triangle's circumcircle. We call a triangle *active* if it is not final.

We created `spdelaunay2d` by modifying an existing Delaunay triangulator so that it keeps in memory only the active triangles and their vertices. This change dramatically reduces the program's memory footprint. The main addition to the triangulator is a component that discovers when active triangles become final, writes them to the output stream, and frees their memory. This component uses a small fraction of the total running time.

Our triangulator maintains two data structures: a triangulation, and a dynamic quadtree that remembers which regions have been finalized. Both are illustrated in Figure 4. The purpose of the quadtree is to identify final triangles, as described in Section 4.2. If the quadtree were fully expanded, its leaves would be the cells of the finalization grid; but there is no need to store the descendants of a quadrant unless it contains both finalized and unfinalized cells. Thus, our quadtree's branches extend and contract dynamically to maintain the finalization state without consuming more memory than necessary.

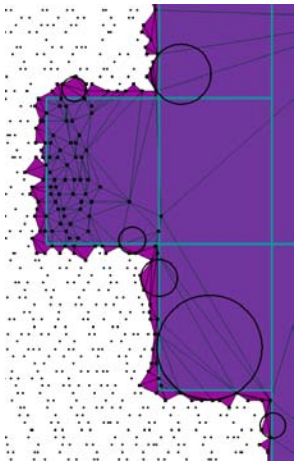

Figure 4: A closeup of streaming Delaunay in 2D. The points on the left have been processed, and their triangles written out. All triangles in this figure are *active*. We have drawn a few representative circumcircles, all of which intersect unfinalized space. At this moment, points are being inserted into the leftmost cell, which will be finalized next.

When `spdelaunay2d` reads a point, it inserts it into the Delaunay triangulation. When it reads a finalization tag, it notes the finalized cell in the quadtree, determines which active triangles become final, writes them to the output stream, and frees their memory. Before a final triangle is written out, any vertex of that triangle that has not yet been output is written out. (Each vertex is delayed in the output stream until the first triangle that depends on it.) After a final triangle is written out, each of its vertices has its memory freed if it is no longer referenced by active triangles.

4.1 Delaunay triangulation with finalization

We use a triangle-based (not edge-based) data structure. Each triangle stores pointers to its three corners and its three neighbors. If a neighboring triangle is final, the corresponding pointer is null.

Standard incremental Delaunay algorithms insert a new point p in two steps. *Point location* finds a triangle whose circumcircle encloses p. *Update* finds all the triangles whose circumcircles enclose p by depth-first search in the triangulation, starting with the triangle where point location ended. These triangles are deleted. New Delaunay triangles adjoining p are constructed, filling the hole formed by the deletion [Bowyer 1981; Watson 1981].

The sequential triangulator that we modified performs point location by walking a straight line from the most recently created triangle toward the new point. (This strategy is advocated by Amenta et al. [2003] in conjunction with the BRIO point reordering performed by our chunker.) In a streaming triangulator, however, this method sometimes fails, because it tries to walk through final triangles, which are no longer in memory. The active triangles do not, in general, define a convex region.

We modified the walking point locator so when it walks into a final triangle (i.e., a null pointer), the walk is restarted from a different starting point. For reasons described in the next section, each leaf of the quadtree maintains a list containing some of the triangles whose circumcircles intersect the leaf's quadrant. We find the quadrant enclosing p and start a new walk from one of the triangles on the quadrant's list. If this walk fails as well, we first try starting from another triangle, and then from triangles on neighboring quadrants' lists, before resorting to an exhaustive search through all the active triangles. In theory we could do better than exhaustive search, but in practice these searches account for an insignificant fraction of our running times. Fewer than 0.001% of point insertions require exhaustive search, and because we retain comparatively few triangles in memory and maintain a linked list of them with the most-recently created triangles at the front of the list, the exhaustive searches are faster than you would expect.

Final triangles pose no problem for the *update* operation. We simply modified the depth-first search so it does not try to follow null pointers. For numerical robustness, we use the robust geometric predicates of Shewchuk [1997] to perform circle tests (deciding whether a circle encloses a point) as well as orientation tests (deciding which side of a line a point lies on). These tests suffice to produce a robust Delaunay triangulator.

4.2 Identifying final triangles

When `spdelaunay2d` reads a finalization tag, it needs to check which active triangles become final—that is, which triangles have circumcircles that no longer touch or enclose an unfinalized cell. We first check whether the circumcircle of a triangle is completely inside the cell that was just finalized—this cheap test certifies many newly created triangles as final. If that test fails, we use the fast circle-rectangle intersection checking code by Shaffer [1990] to test circumcircles against cells. We exploit the quadtree hierarchy to minimize the number of circle-box intersection tests—if a circumcircle does not intersect a quadrant, then it cannot intersect the quadrant's descendants. When it does intersect, we recurse on each child quadrant that intersects the circle.

When a triangle's circumcircle is found to intersect or enclose an unfinalized cell, it would be wasteful to check the triangle again before that cell is finalized. Thus, we link the triangle into a list maintained with the unfinalized cell, and ignore it until the cell's finalization tag arrives (or until a point insertion deletes the triangle). When we check the triangle again, we do not test it against the entire quadtree; we continue searching the quadtree (in preorder traversal) from the cell it is linked with, where the check last failed.

For our algorithm to be correct, circle-box intersection tests cannot report false negatives. False positives are acceptable because they only cause final triangles to stay longer in memory, though we prefer not to have too many of them. Rather than resorting to exact arithmetic (which is slow), we make the intersection tests conservative by computing error bounds E_x, E_y, and E_r on the center coordinates and radius of a triangle's circumcircle. Before we invoke Shaffer's code (or the test if the circle was inside the last finalized cell), we enlarge the box by E_x and E_y and the circle by E_r.

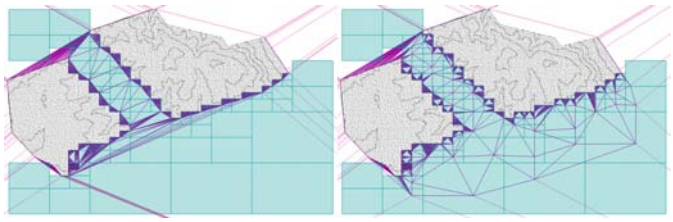

Figure 5: Skinny temporary triangles (left) are avoided by lazily sprinkling one point into each unfinalized quadrant at each level of the evolving quadtree (right).

finalized input points			spdelaunay2d				output mesh
name	# of points file size	options	max active triangles	h:mm:ss disk	 pipe	MB	# of triangles file size
puget (single)	67,125,109 768 MB	l_6	56,163	4:41	5:10	6	(single)
		l_7	48,946	4:23	4:43	7	134,207,228
		l_8	49,316	4:24	4:45	7	2.3 GB
neuse (double)	500,141,313 11.2 GB	l_8e_4	76,337	37:42	39:41	10	(single)
		l_9e_4	60,338	34:27	36:12	10	1,000,282,528
		$l_{10}e_4$	54,802	31:57	33:46	7	16.9 GB
neuse 3×3 (double)	4,501,271,817 101 GB	$l_{10}e_5$	75,081	– : –	5:30:56	11	(single)
		$l_{11}e_5$	67,497	– : –	4:54:40	11	9,002,543,628
		$l_{12}e_5$	68,854	– : –	4:48:47	11	152 GB

Table 2: Performance of `spdelaunay2d` on large terrains. The `spfinalize` option "l_i" selects a quadtree of depth i, and "e_j" finalizes all empty quadrants in the bottom j levels of the tree at the beginning of the stream. Rows list `spdelaunay2d`'s memory footprint (MB) and two timings: one for reading pre-finalized points from *disk*, and one for reading finalized points via a *pipe* from `spfinalize`. Timings and memory footprints do *not* include `spfinalize`, except that the "pipe" timings include `spfinalize`'s third pass, which runs concurrently. For total "pipe" running times, add pass 1 & 2 timings from Table 1. For total "pipe" memory footprints, add the footprint from Table 1. For the corresponding totals in "disk" mode, add the running times of all three finalizer passes, and take the larger memory footprint. Disk timings for the "neuse" tiling are omitted—we do not have enough scratch disk space.

4.3 Effectiveness of reordering with a BRIO

Recall from Section 3 that our finalizer reorders points both locally (in the chunker) and globally (in the sprinkler) to avert the quadratic worst-case behavior that point lattices might produce. Figure 5 depicts snapshots of the triangulator without and with global reordering. Without sprinkling, many very thin, temporary triangles form. Although few of these triangles typically survive to the final triangulation, their circumcircles are large, so they are more likely to be deleted by any given vertex insertion.

In practice we get most of the improvement through local reordering, which lowers the average number of deleted triangles per point insertion from 4.4 to 4.2. Global reordering further lowers this number to 4.1. (Mathematically, the average for points in completely random order would be 4.) The margin increases with the resolution of the finalization grid, but it is not enough to give a measurable improvement in running time on our point streams. Nevertheless, we still prefer the security of knowing that performance will not degenerate on square point lattices in Cartesian order, for which the average number of deleted triangles per point insertion would be proportional to the width of the grid.

4.4 Results

Table 2 summarizes the running times and memory footprints of `spdelaunay2d` for triangulating the largest terrain datasets that we could get our hands on. The measurements were made on a Dell Inspiron 6000D laptop with a 2.13 GHz mobile Pentium processor and 1 GB of memory, running Windows XP. The points were read from a LaCie 5,400 RPM firewire drive with 2 GB/min throughput, and the meshes were written to the 5,400 RPM local disk.

The table shows that while running times and memory footprints vary with the resolution of the finalization grid, little fine tuning is necessary to use our software. Streaming triangulation of 768 MB of single precision floating-point "puget" data takes about six minutes (including preprocessing) for finalization quadtrees of depths 6, 7, and 8. If `spdelaunay2d` is receiving the point stream via a pipe from `spfinalize`, the combined memory footprint of the two programs is 31 MB at depth 6, or 20 MB at depth 8.

We can triangulate the pre-finalized half-billion-point Neuse River Basin in as little as 35 minutes and 10 MB of memory at quadtree depth 9. If the points are not finalized, the finalizer and triangulator together use a combined 70 MB of memory and complete the task in about 48 minutes. This time includes reading 11.2 GB of raw points three times and writing a 16.9 GB streaming mesh. (Our current streaming mesh interface only writes single precision coordinates; in double precision the output would be 24 GB.)

To test our approach's scalability, we modified `spfinalize` to create a 4.5 billion point stream by reading nine translated copies of the Neuse River Basin in a 3×3 tiling. `spfinalize` and `spdelaunay2d` together process 4,501,271,817 points (a number too large to be represented with an unsigned 32-bit integer) with an off-the-shelf laptop. Fewer than seven hours suffice to turn 101 GB of double precision points into a 152 GB terrain composed of 9 billion triangles (double precision output would add 50 GB more). The two programs occupy 166 MB of memory, of which 155 MB is used to chunk the points (see Table 1). The first two passes of `spfinalize` account for nearly two of the seven hours.

Because we do not have enough external storage available, we piped the streaming mesh to a program that measures its size. Alternatively, we can (and do) pipe the output directly into one or more stream processing tools. We can attach a stream module that simplifies the mesh, and another that subsequently extracts isocontours from the simplified mesh. The fact that we produce more data than we can store hints that a common assumption of external memory algorithms, that disk space is for practical purposes unlimited, is not always safe. Whereas most people find their main memory cannot keep up with their disk storage, we have the opposite problem.

5 Streaming 3D Delaunay triangulation

From the stunning performance of streaming Delaunay triangulation in 2D, one would hope for a similar success story for tetrahedralizing points in 3D. Unfortunately, many gigantic data sets in 3D come from scans of surface models, and these are not amenable to a straightforward extension of the finalization procedures we developed for 2D. Delaunay tetrahedra of 3D surface points often have large cirumspheres that touch many cells; only when all touched cells are finalized do such tetrahedra become final. Figure 6 illustrates the 2D analog of this circumstance.

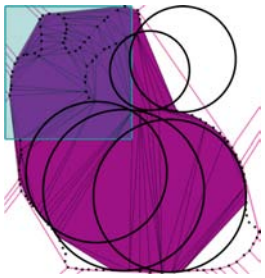

Figure 6: Points sampled on a closed curve. Most of space has been finalized, yet few triangles are final—most circumcircles intersect the unfinalized region.

Nonetheless, we extended `spfinalize` to finalize 3D points with an octree, and implemented a Delaunay tetrahedralizer `spdelaunay3d`, using the same techniques described in Sections 3 and 4. Table 3 shows the performance of `spdelaunay3d` on pre-finalized points on the laptop described in Section 4.4. The "ppm_k" input points consist of every kth vertex of an isosurface mesh extracted from one timestep of a simulation of Richtmyer–Meshkov instability. In this turbulent surface, the points distribute somewhat evenly over a 3D volume and are more suitable for streaming tetrahedralization than surface scans. The table shows that the memory for `spdelaunay3d` is 5–10% of the output size.

Results on two smaller data sets, "sf1" and "f16," appear in Table 4. Each comes from volumetric data used for finite element analysis: points in "sf1" are from a postorder traversal of

finalized input points				spdelaunay3d			output mesh	
name	# of points	MB	opt	max active	h:mm:ss	MB	# tetrahedra	GB
ppm_{16}	11,737,698	136	l_4	951,683	7:42	137	80,751,131	1.4
ppm_8	29,362,621	341	l_5	1,903,241	22:19	306	201,721,882	3.5
ppm_4	58,725,279	686	l_6	4,010,296	56:23	592	405,940,587	7.0
ppm_2	117,450,465	1,422	l_7	6,907,250	2:41:06	795	815,321,347	14

Table 3: Performance of `spdelaunay3d` tetrahedralizing pre-finalized 3D points sampled from the ppm isosurface. The output is a streaming tetrahedral mesh. Option "l_i" indicates that the points are spatially finalized with an octree of depth i. The middle third of the table shows the maximum number of active tetrahedra, the running time (hours:minutes:seconds), and the memory footprint (MB).

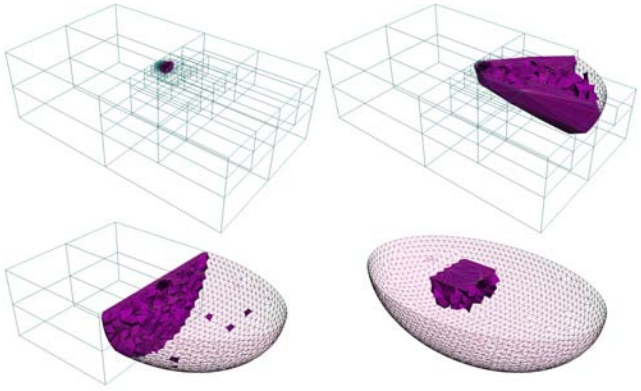

Figure 7: Streaming Delaunay tetrahedralization of the f16 point set. Sprinkle points are turned off for clarity. Most of this model's points are clustered near its center.

an adaptive octree mesh used in CMU's Quake earthquake simulation project. "sf1" is tetrahedralized slowly because its points lie on a grid, often forcing the robust geometric predicates [Shewchuk 1997] to resort to exact arithmetic. The points in "f16" are the vertices of a tetrahedral mesh ordered along a space-filling z-order curve. Figure 7 depicts `spdelaunay3d` as it triangulates "f16."

6 Comparisons

Here we compare the performance of our streaming triangulators with in-core triangulators and with the previous fastest external memory Delaunay triangulator, by Agarwal, Arge, and Yi [2005], which also constructs *constrained* Delaunay triangulations.

Agarwal et al. "process 10 GB of real-life LIDAR data"—the 500 million point Neuse Basin point set (recall Table 2), plus 755,000 segments that constrain the triangulation—"using only 128 MB of main memory in roughly 7.5 hours." This timing omits a preprocessing step that sorts the points along a space-filling Hilbert curve, taking about three additional hours. Their total time is thus 10–11 hours, compared to our 48 minutes to triangulate the unsorted points. This comparison is skewed (in opposite directions) by two differences. First, our triangulator does not read or respect the segments. We plan to add that capability and expect it will cost less than 20% more time for the Neuse data. Second, Agarwal et al. used a slightly faster processor, and much faster disks, than we did.

Our streaming Delaunay triangulators do more work than standard in-core algorithms, because they must identify final Delaunay triangles and tetrahedra. Nevertheless, Table 4 shows that they can outperform state-of-the-art in-core triangulators even for data sets that fit in memory. We compare them with the 2D triangulator Triangle [Shewchuk 1996] and the 3D triangulator Pyramid [Shewchuk 1998], modified to read input points from and write output meshes to the same binary format as our triangulators. Triangle, based on a divide-and-conquer algorithm, is the fastest sequential 2D implementation. Pyramid uses an incremental algorithm.

We used two laptops for our timings to get a sense of when the in-core triangulators start to thrash: a newer laptop described in Section 4.4, with 1 GB of memory, and an older laptop with a 1.1 GHz mobile Pentium III processor and 512 MB of memory. Four data sets appear in Table 4. The 2D data sets, "grbm" and "puget," are described in Section 3 and depicted in Figure 3. The smaller "$puget_5$" is obtained by sampling every fifth point from "puget." The 3D data sets, "f16" and "sf1," are described in Section 5.

input		`spfinalize`		`spdelaunay2d`		total		Triangle			output
name MB	opt	old	MB	old	MB	old	MB	old	−I/O	MB	MB
# of points		new		new		new		new			triangles
grbm 69	l_6	0:04	15	1:07	3	1:11	18	1:47	1:02	495	208
6,016,883		0:04		0:23		0:27		0:34	0:17		12,018,597
$puget_5$ 154	l_6	0:10	7	2:17	4	2:27	11	thrash		863	460
13,423,821		0:10		0:55		1:05		3:45	1:22		26,840,720

input		`spfinalize`		`spdelaunay3d`		total		Pyramid			output
name MB	opt	old	MB	old	MB	old	MB	old	−I/O	MB	MB
# of points		new		new		new		new			tetrahedra
f16 13	l_9m_5	0:01	5	1:16	28	1:17	33	2:53	2:46	262	125
1,124,648		0:01		0:34		0:35		1:37	1:26		7,027,642
sf1 29	l_6m_5	0:02	16	9:57	29	9:59	45	thrash		537	251
2,461,694		0:02		4:15		4:17		5:16	4:57		13,980,309

Table 4: Running times (minutes:seconds) and memory footprints (MB) of triangulators on an old laptop (top of each time box) with 512 MB memory and a new laptop (bottom of each time box) with 1 GB memory, for several 2D and 3D point sets. `spfinalize` pipes its output to `spdelaunay2d` or `spdelaunay3d`; timings for `spfinalize` reflect only the first two passes over the input stream, and timings for `spdelaunay2d` or `spdelaunay3d` reflect the combined times for the triangulator and the finalizer's third pass. The "total" column lists the start-to-end running time and memory footprint of the triangulation pipe. For the in-core triangulators Triangle and Pyramid, we report total running time and the running time excluding I/O ("−I/O"). Option "m_5" means subtrees with less than 5K points are collapsed into their root cell.

The most striking differences are the memory footprints. `spdelaunay2d` uses less than 1% of the space of Triangle; `spdelaunay2d` and `spfinalize` together use less than 5%. `spdelaunay3d` uses less than 11% of the space of Pyramid; `spdelaunay3d` and `spfinalize` together use less than 13%. Moreover, Triangle and Pyramid's memory footprints increase linearly with the size of the triangulation, whereas the streaming triangulators' memory footprints increase more slowly with the stream size. Of course, the in-core triangulators begin thrashing long before the streaming triangulators would. Triangle begins to thrash on the new laptop at about 14 million input points. Compare this with the 4.5 billion points we have triangulated by streaming.

The running times are more surprising. How can the streaming triangulators, with the extra work of finalization, run faster than dedicated in-core triangulators? First, they offset the extra work by overlapping computation with file I/O, whereas Triangle and Pyramid do not. The speed of the streaming triangulators on pre-finalized points is almost entirely CPU-bound. If `spdelaunay2d`, while triangulating the Neuse Basin point stream (recall Table 2), discards the 16.9 GB output mesh stream instead of writing it to disk, it saves only three minutes of the 35-minute processing time.

Second, the streaming triangulators benefit from improved cache performance because of their much smaller memory footprints.

7 Conclusions

Researchers with whom we have discussed out-of-core Delaunay triangulation suggest, almost by reflex, sorting the points first. For data sets with no spatial coherence at all, we too advocate sorting. But in our experience, large, real-world data sets have plenty of spatial coherence. The power of exploiting that spatial coherence is perhaps best illustrated by two facts. First, it takes Agarwal et al. [2005] three hours to Hilbert sort the same point set we triangulate in 48 minutes. Second, our triangulator runs as quickly on the original Neuse point data as on the Hilbert-sorted Neuse points, which were both kindly provided by Agarwal et al. [2005].

We realize the benefits of sorting, at much less cost, by docu-

menting the existing spatial coherence with *spatial finalization* and enhancing it by reordering points. In analogy to aikido, we use the data's spatial coherence to control and direct the data with small efforts, rather than fight it head on (by sorting it). One advantage is speed. Another advantage is that we can visualize the early output of a pipeline of streaming modules soon after starting it.

We have described just one method of spatial finalization for point sets. We choose a depth-k quadtree/octree because we can describe it succinctly with a bounding box and integer k, and it is relatively simple to determine which cells a sphere intersects. We believe it is possible to eliminate the first pass of `spfinalize` by computing a quadtree/octree partitioning adaptively—without advance knowledge of the bounding box—during the second pass. Binary space partitions, k-d trees, and many other spatial subdivisions would work too. If a point stream is sorted along a space-filling curve like a Hilbert or z-order curve, the stream is chunked, and finalization can be implicit—a cell is finalized when the next point leaves it. Sweepline algorithms, such as Fortune's [1992] for Delaunay triangulation, generate a point stream with implicit spatial finalization: they sort the points by one coordinate, thereby partitioning the plane into slabs. At each event, they finalize a slab, and could potentially produce partial output and free data structures. Likewise, Pajarola's [2005] streaming k-neighbor computation finalizes slabs of space with a sweep plane. But these methods bring with them the disadvantages of sorting discussed above.

The Achilles' heel of our 3D streaming triangulator is that it performs poorly on point clouds sampled from surfaces. The Delaunay triangulations of these point clouds have many tetrahedra with large circumspheres, which intersect many cells and are thus long-lived. We believe this problem can be solved by using more sophisticated, non-disjoint finalization regions computed by a randomized divide-and-conquer technique of Clarkson [1988]. Clarkson's method covers space with overlapping spherical regions tailored to the point set, and guarantees that each Delaunay circumsphere is covered by a constant number of these regions; yet no region contains too many points. (Agarwal et al. use the same random sampling technique to divide a constrained Delaunay triangulation into subproblems. We propose to use it for spatial finalization instead.)

Implementations of traditional 2D divide-and-conquer Delaunay algorithms [Shamos and Hoey 1975] are faster than incremental implementations, and even run in expected linear time on random points from some distributions [Katajainen and Koppinen 1988]. 2D divide-and-conquer algorithms seem amenable to a streaming implementation using our spatial finalization method. The key to fast streaming is to merge adjacent triangulations in an order dictated by the data, instead of an *a priori* order. Unfortunately, this rules out the best-known generalization of the divide-and-conquer approach to dimensions above two, the Delaunay Wall algorithm [Cignoni et al. 1998], which constructs tetrahedra in an inflexible, predetermined order. We do not know how to create a 3D streaming divide-and-conquer Delaunay algorithm.

As huge data sets become ubiquitous in geometry processing, we hope that streaming geometry with finalization information and low width will become common. If point-creating programs would include finalization tags in their output streams, we could pipe them directly to our Delaunay triangulators, and begin producing triangles or tetrahedra even before all the points are created. The advantages of stream processing are so strong that we believe the producers of huge geometric data sets will have a profound incentive to make the modest efforts required to improve their spatial coherence and include finalization information.

Acknowledgments. We thank Kevin Yi for supplying the Neuse Basin data and extensive information about his work [Agarwal et al. 2005]. This work was supported by National Science Foundation Award CCF-0430065, National Geospatial-Intelligence Agency/Defense Advanced Research Projects Agency Award HM1582-05-2-0003, and an Alfred P. Sloan Research Fellowship.

References

AGARWAL, P. K., ARGE, L., AND YI, K. 2005. I/O-efficient construction of constrained Delaunay triangulations. In *Proceedings of the Thirteenth European Symposium on Algorithms*, 355–366.

AMENTA, N., CHOI, S., AND ROTE, G. 2003. Incremental constructions con BRIO. In *Proceedings of the Nineteenth Annual Symposium on Computational Geometry*, Association for Computing Machinery, San Diego, California, 211–219.

BLANDFORD, D. K., BLELLOCH, G. E., CARDOZE, D. E., AND KADOW, C. 2005. Compact representations of simplicial meshes in two and three dimensions. *International Journal of Computational Geometry and Applications 15*, 1 (Feb.), 3–24.

BLELLOCH, G. E., HARDWICK, J. C., MILLER, G. L., AND TALMOR, D. 1999. Design and implementation of a practical parallel Delaunay algorithm. *Algorithmica 24*, 3–4 (Aug.), 243–269.

BOWYER, A. 1981. Computing Dirichlet tessellations. *Computer Journal 24*, 2, 162–166.

BUCHIN, K. 2005. Constructing Delaunay triangulations along space-filling curves. In *Second Symposium on Voronoi Diagrams*, 184–195.

CIGNONI, P., MONTANI, C., AND SCOPIGNO, R. 1998. DeWall: A fast divide and conquer Delaunay triangulation algorithm in E^d. *Computer-Aided Design 30*, 5 (Apr.), 333–341.

CLARKSON, K. L., AND SHOR, P. W. 1989. Applications of random sampling in computational geometry, II. *Discrete & Computational Geometry 4*, 1, 387–421.

CLARKSON, K. L. 1988. A randomized algorithm for closest-point queries. *SIAM Journal on Computing 17*, 4 (Aug.), 830–847.

FORTUNE, S. 1992. Voronoi diagrams and Delaunay triangulations. In *Computing in Euclidean Geometry*, D.-Z. Du and F. Hwang, Eds., vol. 1 of *Lecture Notes Series on Computing*. World Scientific, Singapore, 193–233.

FREDERICK, C. O., WONG, Y. C., AND EDGE, F. W. 1970. Two-dimensional automatic mesh generation for structural analysis. *International Journal for Numerical Methods in Engineering 2*, 133–144.

ISENBURG, M., AND GUMHOLD, S. 2003. Out-of-core compression for gigantic polygon meshes. *ACM Transactions on Graphics 22*, 3 (July), 935–942.

ISENBURG, M., AND LINDSTROM, P. 2005. Streaming meshes. In *Visualization '05 Proceedings*, 231–238.

ISENBURG, M., LINDSTROM, P., GUMHOLD, S., AND SNOEYINK, J. 2003. Large mesh simplification using processing sequences. In *Visualization '03*, 465–472.

KARP, R. M. 1992. On-line algorithms versus off-line algorithms: How much is it worth to know the future? In *IFIP 12th World Computer Congress*, North-Holland, J. van Leeuwen, Ed., vol. A-12 of *IFIP Transactions*, 416–429.

KATAJAINEN, J., AND KOPPINEN, M. 1988. Constructing Delaunay triangulations by merging buckets in quadtree order. *Fundamenta Informaticae XI*, 11, 275–288.

KUMAR, P. 2003. Cache oblivious algorithms. In *Algorithms for Memory Hierarchies, LNCS 2625*, U. Meyer, P. Sanders, and J. Sibeyn, Eds. Springer-Verlag, 193–212.

LAWSON, C. L. 1977. Software for C^1 surface interpolation. In *Mathematical Software III*, J. R. Rice, Ed. Academic Press, New York, 161–194.

LIU, Y., AND SNOEYINK, J. 2005. A comparison of five implementations of 3D Delaunay tessellation. In *Combinatorial and Computational Geometry*, vol. 52 of *MSRI Publications*. Cambridge, 435–453.

OKABE, A., BOOTS, B., SUGIHARA, K., AND CHIU, S. N. 2000. *Spatial tessellations: Concepts and applications of Voronoi diagrams*, 2nd ed. Wiley, New York.

PAJAROLA, R. 2005. Stream-processing points. In *Visualization '05 Proc.*, 239–246.

QUILLIN, M., 2002. Flood plain maps better, but late—years late, March 11. Raleigh News & Observer.

SHAFFER, C. A. 1990. Fast circle-rectangle intersection checking. In *Graphics Gems*. Academic Press Professional, Inc., San Diego, CA, USA, 51–53.

SHAMOS, M. I., AND HOEY, D. 1975. Closest-point problems. In *16th Annual Symposium on Foundations of Computer Science*, IEEE Press, 151–162.

SHEWCHUK, J. R. 1996. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, M. C. Lin and D. Manocha, Eds., vol. 1148 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, May, 203–222.

SHEWCHUK, J. R. 1997. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry 18*, 3 (Oct.), 305–363.

SHEWCHUK, J. R. 1998. Tetrahedral mesh generation by Delaunay refinement. In *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*, Association for Computing Machinery, Minneapolis, Minnesota, 86–95.

SU, P., AND DRYSDALE, R. L. S. 1995. A comparison of sequential Delaunay triangulation algorithms. In *Proceedings of the Eleventh Annual Symposium on Computational Geometry*, 61–70.

VITTER, J. S. 2001. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys 33*, 2, 209–271.

WATSON, D. F. 1981. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *Computer Journal 24*, 2, 167–172.

YOON, S., LINDSTROM, P., PASCUCCI, V., AND MANOCHA, D. 2005. Cache-oblivious mesh layouts. *ACM Transactions on Graphics 24*, 3 (July), 886–893.